COMP251 Winter 2018

Wenzong Xia

September 27, 2018

Contents

Prerequisite Materials

Time Complexity

In computer science, the *time complexity* is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.

Big-O

Big-O ($O()$) is one of five standard asymptotic notations. In practice, it is used as a *loose upper-bound*.

Definition. — *(Big-O, $O()$): Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there exists a real constant $c > 0$ and there exists an integer constant $n_0 > 1$ such that $f(n) \leq c * g(n)$ for every integer $n \geq n_0$.*

Little-O

Little-o $o()$ is used to describe an upper-bound that cannot be tight.

Definition. — *(Little-o, $o()$): Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $o(g(n))$ if for any real constant $c > 0$, there exists an integer constant $n_0 > 1$ such that $f(n) < c * g(n)$ for every integer $n \geq n_0$.*

Big-Omega

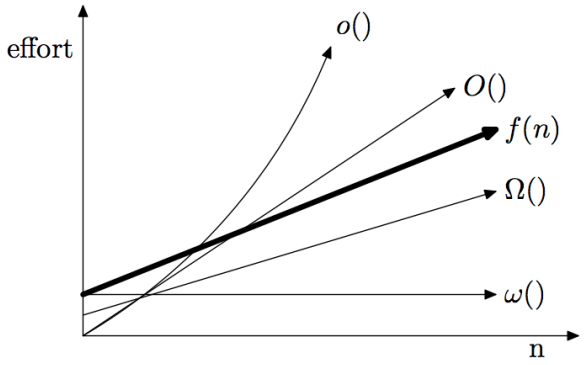
Big-Omega $\Omega()$ is the tight lower bound notation.

Definition. — *(Big-Omega, $\Omega()$): Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $\Omega(g(n))$ if there exists a real constant $c > 0$ and there exists an integer constant $n_0 \geq 1$ such that $f(n) \geq c * g(n)$ for every integer $n > n_0$.*

Little-Omega

Little-Omega, $\omega()$ describes the loose lower bound.

Definition. — *(Little-Omega, $\omega()$): Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $\omega(g(n))$ if for any real constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $f(n) > c * g(n)$ for every integer $n \geq n_0$.*

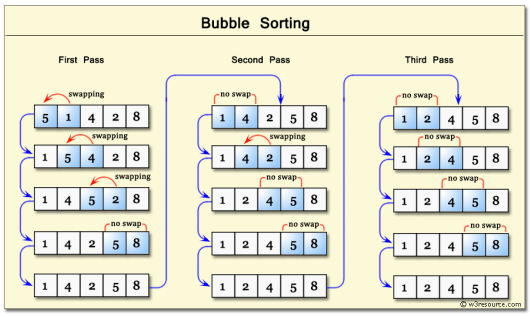


Fundamental Algorithms

BubbleSort Algorithms

Algorithm 0.0.1: BubbleSort Algorithm

Result: BubbleSort$(x_1, \cdots, x_n)$
$counter = 0$;
continue=true;
$n = length(S)$;
while $continue = true$ **do**
  continue=false;
  for $i = 1$ *to* $n - 1$ **do**
    if $A[i-1] > A[i]$ **then**
      swap$(A[i-1], A[i])$;
      swapped = true;
  end
  $counter++$;
end

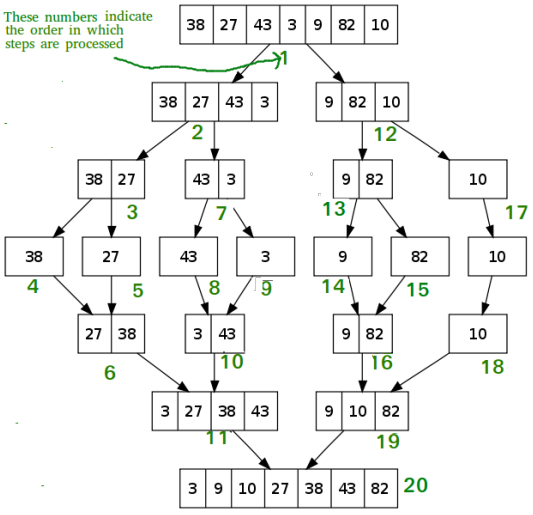


Remark. The runtime of the BubbleSort Algorithm is $O(n^2)$.

MergeSort Algorithm

Algorithm 0.0.2: MergeSort Algorithm

Result: MergeSort$(x_1, \cdots, x_n)$
if $n = 1$ **then**
  output x_1;
else
  output Merge{MergeSort$(x_1, \cdots, x_{\lfloor \frac{n}{2} \rfloor})$, MergeSort$(x_{\lfloor \frac{n}{2}+1 \rfloor}, \cdots, x_n)$};
end

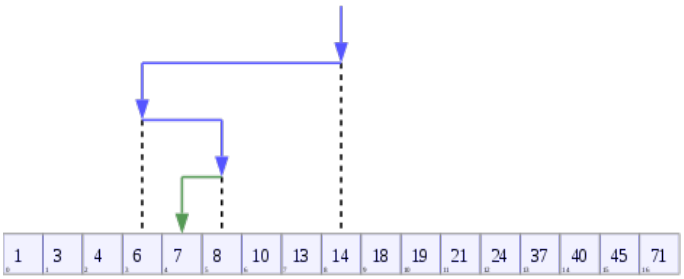


Remark. The runtime of the BubbleSort Algorithm is $O(n^2)$.

BinarySearch Algorithm

Algorithm 0.0.3: BinarySearch Algorithm

Result: BinarySearch$(a_1, \cdots, a_n)$; k
while $n > 0$ **do**
  if $a_{\lceil \frac{n}{2} \rceil} = k$ **then**
    Output YES;
  else if $a_{\lceil \frac{n}{2} \rceil} > k$ **then**
    Output BinarySearch$(a_1, \cdots, a_{\lceil \frac{n}{2} \rceil - 1})$; k;
  else if $a_{\lceil \frac{n}{2} \rceil} < k$ **then**
    Output BinarySearch$(a_{\lceil \frac{n}{2} \rceil - 1}), \cdots, a_{n-1}, an)$; k;
end
Output NO;



Remark. The runtime of the BinarySearch Algorithm is $O(n \log n)$.

Graph Theory

Definition. — ***Undirected Graphs***: *An undirected graph $G = (V, E)$ consists of:*

- *A set V of vertices (or nodes).*
- *A set E of edges (or links) denoting unordered vertex pairs.*
- *We set $n = |V|$ to be the cardinality of the vertex set.*
- *We set $n = |E|$ to be the cardinality of the edge set.*

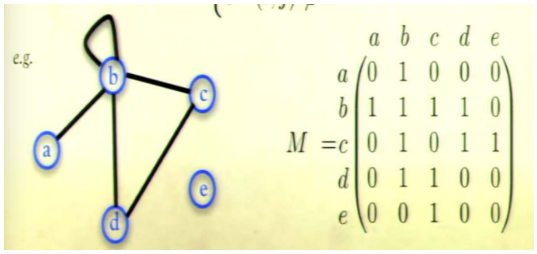
Definition. — ***Directed Graphs***: *An directed graph $G = (V, E)$ consists of:*

- *A set V of vertices (or nodes).*
- *A set A of **arcs** (directed edges) denoting ordered vertex pairs.*
- *We set $n = |V|$ to be the cardinality of the vertex set.*
- *We set $n = |A|$ to be the cardinality of the arc set.*

Definition. — ***Adjacency Matrix(undirected graphs)***: *For an indirected graph, an adjacency matrix M has the properties that:*

- *There is a **row** for each vertex.*
- *There is **column** for each vertex.*
- *The $ij-th$ entry of the matrix is defined by:*

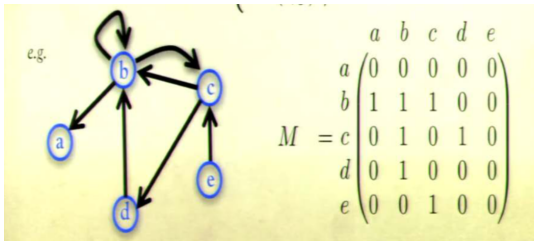
$$M_{ij} = \begin{cases} 1 (i,j) \in E \\ 0 (i,j) \notin E \end{cases}$$



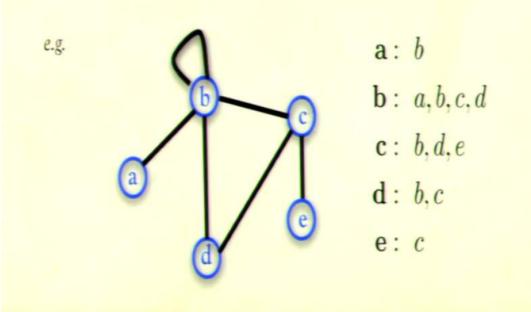
Definition. — ***Adjacency Matrix(directed graphs)***: *For an directed graph, an adjacency matrix M has the properties that:*

- *There is a **row** for each vertex.*
- *There is **column** for each vertex.*
- *The $ij-th$ entry of the matrix is defined by:*

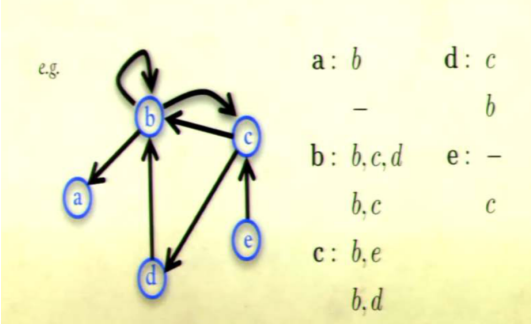
$$M_{ij} = \begin{cases} 1 (i,j) \in A \\ 0 (i,j) \notin A \end{cases}$$



Definition. — ***Adjacency List(undirected graph)****: An undirected graph can be stored using adjacency lists. For each vertex i, we store a list of the* ***neighbors*** *of i.*



Definition. — ***Adjacency List(directed graph)****: A directed graph can be stored using adjacency lists. For each vertex i, we store a list of the* ***in-neighbors*** *of i. For each vertex i, we also store a list of the* ***out-neighbors*** *of i. Equivalently, we store a list of the edges* ***incident*** *to each vertex.*



Remark. Adjacency Lists versus Adjacency Matrices: The main difference is in the amount of space required.

- An adjacency matrix requires storing $\omega(n^2)$ numbers.
- An adjacency list requires storing $\omega(m)$ numbers.
- In any graph $m = O(n^2)$ and often $m \leq n^2$. $\Rightarrow$In **sparse graphs**, it is much more preferable to use adjacency lists.

Graph Structures

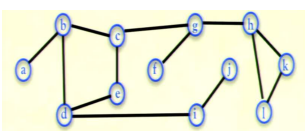
Definition. — ***Walks****: a walk is a list of vertices $\{v_0, v_1, v_2, \cdots, v_l\}$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq 1 \leq l$.*

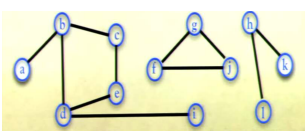
Definition. — ***Circuits****: a circuit is a walk $\{v_0, v_1, v_2, \cdots, v_l\}$ where $v_0 = v_l$. (A circuit is a closed walk). An* ***Eulerian Circuit*** *is a circuit that uses every edge exactly once.*

Definition. — ***Paths****: a path is a walk where every vertex is distinct.*

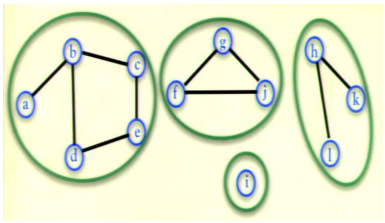
Definition. — ***Cycles****: a cycle is a walk $\{v_0, v_1, v_2, \cdots, v_l\}$where every vertex is distinct. except for the end-vertices $v_0 = v_l$. An* ***Hamilton Cycle*** *is a cycle that uses every vertex exactly once.*

Definition. — *A graph is* ***connected*** *if for every pair of vertices $u, v \in V$, it is possible to walk from v to u.*
A graph is ***disconnected*** *if there exists a pair of vertices $u, v \in V$, for which there is no possible walk from u to v.*



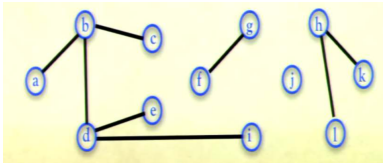


Definition. — ***Graph Components***: *A connected subgraphs are called the components of the graph. Thus a connected graph has exactly one component.*

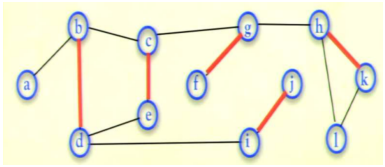


Trees

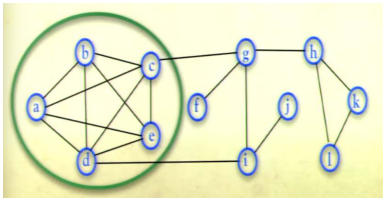
Definition. — ***Tree***: *A tree is a connected component with no cycles.* ***Forest***: *A forest is a graph whose components are all trees. A tree is* ***spanning*** *if it contains every vertex in the graph.*



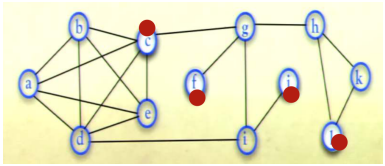
Definition. — ***Matching***: *A matching is a set of vertex-disjoint edges. Hence, each vertex is incident to at most one edge in the matching. A matching is* ***perfect*** *if every vertex is incident to an edge in the matching.*



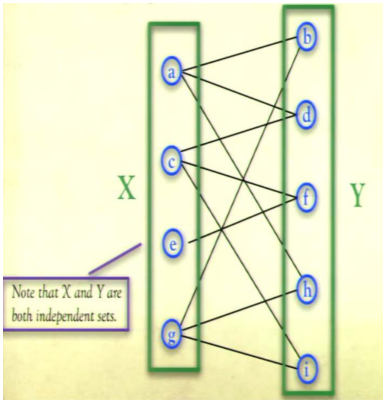
Definition. — ***Cliques***: *A clique is a set of pairwise adjacent vertices.*



Definition. — ***Independent Sets***: *An independent sets (stable set) is a set of pairwise non-adjacent vertices.*



Definition. — ***Bipartite Graphs***: *In a bipartite graph, the vertex set can be partitioned as $V = X \cup Y$ such that every edge has one end-vertex in X and one end-vertex in Y.*



Some Theorems on Undirected Graphs

Lemma 0.0.1. Let $\Gamma(v) = \{u : (u, v) \in E\}$ be the set of neighbors of v. The degree, $deg(v)$, of a vertex v is a cardinality of $\Gamma(v)$. The **Handshaking Lemma**: In an undirected graph, there are an even number of vertices with odd degree.

Proof. We have:

$$\begin{aligned} 2 \cdot |E| &= \sum_{v \in V} deg(v) \\ &= \sum_{v \in O} deg(v) + \sum_{v \in \varepsilon} deg(v) \\ &= \sum_{v \in O} deg(v) = 2 \cdot |E| - \sum_{v \in \varepsilon} deg(v) \end{aligned}$$

□

Theorem 0.1. — ***Euler's Theorem****: An undirected graph contains an Euler Circuit if and only if every vertex has even edges. (Euler Circuit: a path starting and ending on the same vertex which visits each edge exactly once).*

Theorem on Trees

Definition. — ***Leaves****: A vertex with degree one in a tree is called a leaf.*

Theorem 0.2. — *A tree with n vertices has $n - 1$ edges.*

Theorem 0.3. — ***Hall's Condition****: $\forall B \subseteq X$, $|\Gamma(B)| \geq |B|$.* ***Hall's Theorem****: A bipartite graph, with $|X| = |Y|$, contains a perfect matching if and only if Hall's Condition is satisfied.*

Breadth First Search

Generic Search Algorithm

Algorithm 0.0.4: Generic Search Algorithm

```
Result: Search Algorithm(∗, r)
Put (∗, r) into a bag;
while the bag is not empty do
    Remove (u, v) from the bag;
    if v is unmarked then
        Mark v;
        Set p(v) ← u (set u to be the predecessor of v);
        for each arc (v, u) do
            Put (v, u) into the bag;
        end
end
```

Remark. We search each arc out of v only once, when v is first marked. The arc is then added to the bag once and later removed from the bag. Hence, the runtime of the algorithm is:

$$\text{Runtime} = O(m). \text{ It is a } \textbf{linear time} \text{ algorithm.}$$

Theorem 0.4. — *Let G be a connected, undirected graph. Then the search algorithm find every vertex in G.*

Proof. We must show that each vertex is marked by the search algorithm. We prove this by induction on the smallest number of k of edges in a path from the vertex to the root.
Base Case: $k = 0$. Then v is the root vertex r. But r is the first vertex marked in the algorithm, so the base case holds.
Induction Hypothesis: Assume any vertex v has a path of $k-1$ (or fewer) edges to the root r is marked.
Induction Step: Assume there is a path P with k edges from v to r. Specifically let $P = \{v = v_k, v_{k-1}, \cdots, v_1, v_0 = r\}$. Thus, there is a path Q with $k-1$ edges from $u = v_{k-1}$ to r. That is, $Q = \{u = v_{k-1}. \cdots, v_1, v_0 = r\}$. So, by the induction hypothesis, the vertex u is marked. After we mark u, we place the edges incident to it in the bag. $\Rightarrow$ The edge (u, v) is added to the bag. Thus, when (u, v) is removed from the bag we will mark v (if it is not already marked). □

For directed graphs, a similar argument proves that each vertex that has a directed path to it from the root is marked.

Search Trees

Observe that each non-root vertex has exactly one predecessor.

Theorem 0.5. — *Let G be a connected, undirected graph. Then the predecessor edges from a tree rooted at r.*

Proof. By induction on the number k of marked vertices.
Base Case: $k = 1$. The root vertex r is the first vertex marked. $\Rightarrow$ Trivially, we have a tree rooted at r, so the base case holds.
Induction Hypothesis: Assume the predecessor edges for the first $k-1$ marked vertices from a tree rooted at r.
Induction Step: Let v be the $k-th$ vertex to be marked. Assume v was marked when we remove the edge (u, v). $\Rightarrow u = P(v)$. But (u, v) was added to the bag when we marked vertex u. $\Rightarrow$ Vertex u is in the set S of the first $k-1$ vertices to be marked. by induction hypothesis, the predecessor edges for S form a tree T rooted at r. $\Rightarrow T \cup (p(v), v)$ is a tree rooted at r in the first k marked vertices. □

Remark. There is some flexibility in how to implement the search algorithm. If the bag contains many arcs, which one should we remove? In fact, in computer science, the "bag" is just short-hand for a data structure.

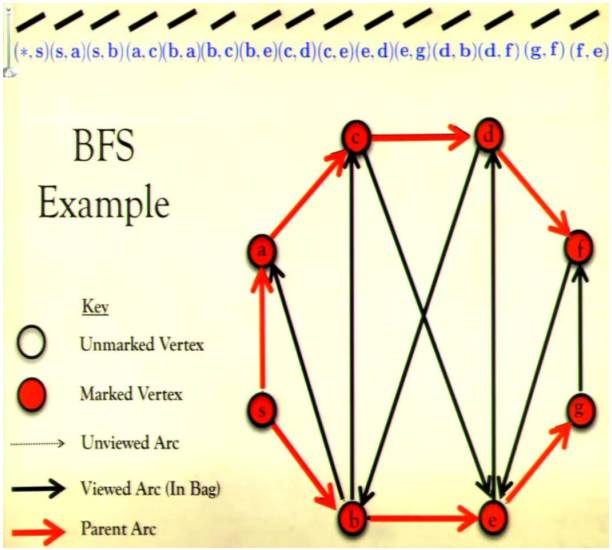
Queue $\Rightarrow$ *Breadth First Search*
Stack $\Rightarrow$ *Depth First Search*
Priority Queue $\Rightarrow$ *Minimum Spanning Tree Algorithm.*

Breadth First Search (BFS)

Using a **Queue(FIFO)** data structure produces:

Algorithm 0.0.5: Breadth First Search Algorithm

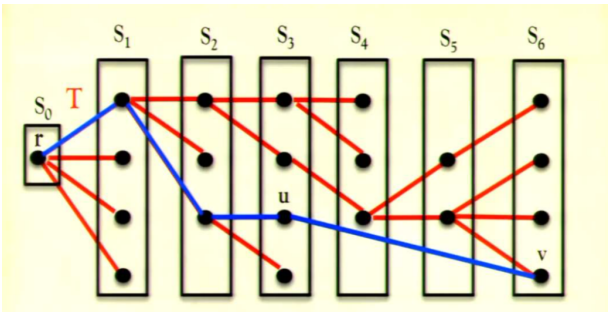
```
Result: Breadth First Search Algorithm(∗, r)
Put (∗, r) into a Queue;
while the Queue is not empty do
    Remove (u, v) from the Queue;
    if v is unmarked then
        Mark v;
        Set p(v) ← u (set u to be the predecessor of v);
        for each arc (v, u) do
            Put (v, u) into the Queue;
        end
end
```



Breadth First Search Trees

Since BFS uses queue, it is easy to verify that the edges are added to the queue in order of their distance from r and the vertices are marked in order of their distance from r. Specifically:

Theorem 0.6. — *For any vertex v, the path from v to r given by the search tree T of predecessor edges is a shortest path.*



Let S_l be the set of "layer" of vertices at distance l from r in T. Then a vertex $v \in S_l$ is also at distance l from r in the whole graph G. Thus implies for every non-tree edge (u, v), u and v are either in the same layer of in adjacent layers. If not, we can find a shorter path to the root from u to v.

BFS and Bipartite Graphs

Recall a graph $G = (V, E)$ is bipartite if $V = U \cup Y$ and every edge has exactly one end-point in X and one end-point in Y.

Theorem 0.7. — *A graph G is bipartite if and only if it contains no add length cycles.*

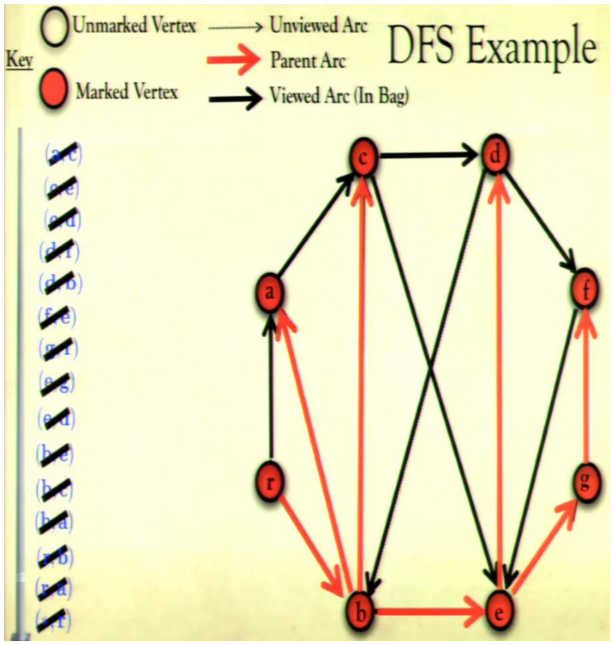
Proof. Assume G contains as a subgraph an odd length cycle C. Let $C = \{v_0, v_1, v_2, \cdot, v_{2k}\}$. WLOG we may assume vertex $v_0 \in Y$. Therefore: $v_0 \in Y \Rightarrow$ But $(v_0, v_{2k}) \in E$. Thus, $v_0 \in Y \Rightarrow v_1 \in X \Rightarrow v_{2k} \in Y$. This contradiction proves that G contains no odd cycle. □

Depth First Search (DFS)

Using a stack(LIFO) data structure produces:

Algorithm 0.0.6: Breadth First Search Algorithm

Result: Breadth First Search Algorithm$(*, r)$
Put $(*, r)$ into a Stack;
while *the Stack is not empty* **do**
  Remove (u, v) from the Stack;
  if v *is unmarked* **then**
    Mark v;
    Set $p(v) \leftarrow u$ (set u to be the predecessor of v);
    for *each arc* (v, w) **do**
      Put (v, w) into the Stack;
    end
end



Recall, we proved that the search algorithm produces a search tree.

Theorem 0.8. — *Let G be a connected, undirected graph. Then the predecessor edges form a tree rooted at r. However, the structure of the DFS tree is quite different from that of a BFS tree.*

Algorithm 0.0.7: Breadth First Search Algorithm

Result: Depth First Search Algorithm(r)
Mark r;
for *each edge* (r, v) **do**
  if v *is unmarked* **then**
    Set $p(v) \leftarrow r$ (keep track of the predecessor);
    RecursiveDFS(v);
end

Theorem 0.9. — *Let T be a DFS tree in an undirected graph G. Then, for every edge (u, v), either u is an ancestor of v in T or v is an ancestor of u.*

Proof. WLOG assume u is marked before v. Consider the time u is marked during RecursiveDFS(u). In RecursiveDFS(u) the algorithm examines each arc incident to u.
Case I: v is unmarked when Recursive(u) examines (u, v). Then RecursiveDFS(u) sets $P(v) \leftarrow u \Rightarrow (u, v)$ is a tree edge.
Case II: v is marked when RecursiveDFS(u) examines (u, v). But v was marked after u, so it was marked during RecursiveDFS(u). Thus, we have a series of vertices $\{u = w_0, w_1, \cdots, w_{l-1}, w_l = v\}$ where $p(w_k) = w_{k-1} \Rightarrow u$ is an ancestor of v in T. $\Rightarrow (u, v)$is a back edge. □

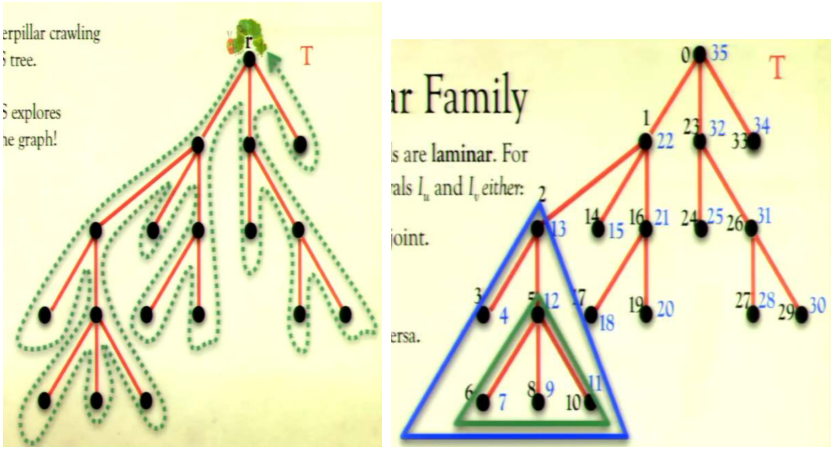
Corollary 1. — *Let T be a DFS tree in an undirected graph G. Then, every non-tree edge is a back edge.*

Pre-visit and Post-visit

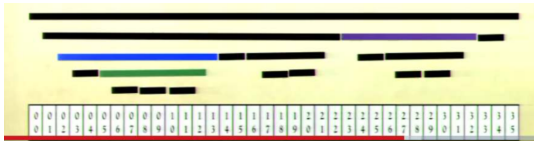
We can add a clock to record when we visit a vertex for the first(previsit) and last time(postvisit).

Caterpillar Crawls

The way DFS explores the vertices of the graph is like this; $Pre(v)$ is the time the caterpillar arrives at the subtree rooted at vertex v. $Post(v)$ is the time the caterpillar exits the subtree rooted at vertex v.



We can represent each vertex by an interval over the real numbers.



To be continued...

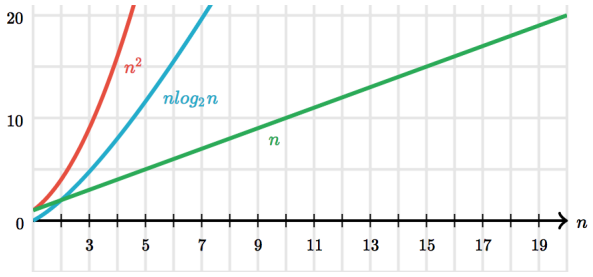
Introduction

Good vs Bad Algorithms

Theorem 0.10. — *(Central Paradigm of CS). An algorithm A is good if A runs in **polynomial time** in the input size n. In other words, A runs in time $T(n) = O(n^k)$ for some constant k.*
Remark. "A runs in polynomial time in the input size n" is equivalent to the "input size that A can solve, in a fixed amount T of time, scales multiplicity with increasing computational power".

We say that an algorithm is bad if it runs in **exponential time**. For example: considering the problem of sorting n numbers:

- A Good Algorithm: **MergeSort** runs in time $O(n * logn)$ (roughly linear).
- A Bad Algorithm: **BruteForce Search** runs in time $O(n * n!)$.



Software vs Hardware

Improvement in hardware will never overcome bad algorithm design. Indeed, current dramatic breakthroughs in CS are based upon better (faster and higher performance) algorithm techniques.

Part I

Recursive Algorithm

1 | Divide + Conquer Algorithms

Solving a problem by reducing it (or a subproblem of it) to another problem is the most fundamental technique in algorithm design. Algorithm A use another Algorithm B as a sub-routine. There are numerous advantages of doing so:

1. Code Verification: Correctness of A is independent of B.
2. Code Reuse: A great time saver.

Theorem 1.0.1: Divide and Conquer Algorithm

Divide and Conquer Algorithm recursively breaks up a problem of size n in smaller sub-problems such that:

- There are exactly a sub-problems.
- Each sub-problem has size at most $\frac{1}{b} * n$
- Once solved, the solutions to the sub-problems can be combined to produce a solution to the original problem in time $O(n^d)$

Run time of a divide and conquer algorithm satisfies the recurrence:

$$T(n) = a * T(\frac{n}{b}) + O(n^d)$$

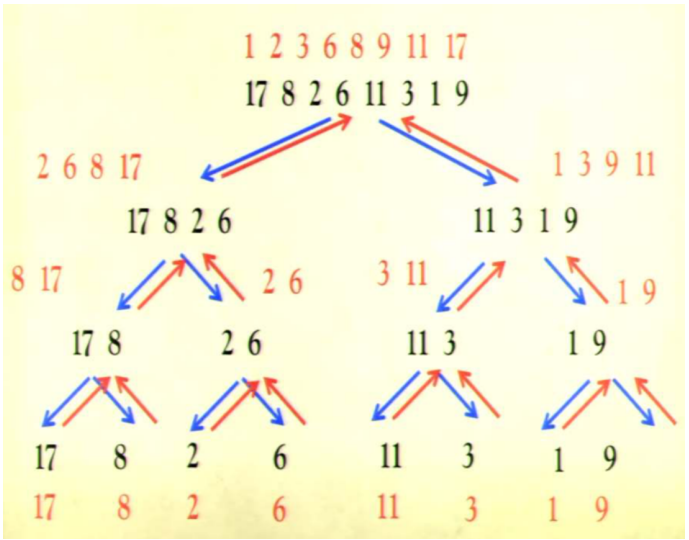
1.1 MergeSort

Here, we want to sort n numbers into non-decreasing order.

Algorithm 1.1.1: MergeSort Algorithm

Result: MergeSort$(x_1, \cdots, x_n)$
if $n = 1$ **then**
  output x_1;
else
  output Merge{MergeSort$(x_1, \cdots, x_{\lfloor \frac{n}{2} \rfloor})$, MergeSort$(x_{\lfloor \frac{n}{2}+1 \rfloor}, \cdots, x_n)$};
end

Example 1.1. — *MergeSort example*



Remark. MergeSort does work: The division terminates with a set of base case of size 1. Using strong induction, and given the validity of the Merge step, we are done. (MergeSort trivially works on base case).

Theorem 1.1. — *MergeSort runs in time* $O(n \cdot \log n)$

Proof. First, MergeSort algorithm has the form of:

$$T(n) = 2T(\frac{n}{2}) + cn$$

where the first term is due to recursion on 2 problems with half the size and the second term is the linear time to merge 2 sorted lists. Also the base case is $T(n) = 1$. Now, bu adding the **dummy numbers**, we may assume that n is a power of two; $n = 2^k$. So we have:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + cn \\ &= 2(2T(\frac{n}{4}) + \frac{cn}{2}) + cn \\ &= 2^2T(\frac{n}{4}) + 2cn \\ &= 2^3T(\frac{n}{8}) + 3cn \\ &= \cdots \\ &= 2^kT(1) + kcn \\ &= 2^k + kcn \\ &= n(1 + kc) \end{aligned}$$

So, $T(n) = O(nk) = O(n \log n)$ □

1.2 Binary Search

We can search for a key k in a sorted array list of cardinality n by using the binary search algorithm.

Algorithm 1.2.1: BinarySearch Algorithm

Result: BinarySearch($a_1, \cdots, a_n$); k
while $n > 0$ **do**
  if $a_{\lceil \frac{n}{2} \rceil} = k$ **then**
    Output YES;
  else if $a_{\lceil \frac{n}{2} \rceil} > k$ **then**
    Output BinarySearch($a_1, \cdots, a_{\lceil \frac{n}{2} \rceil - 1}$); k;
  else if $a_{\lceil \frac{n}{2} \rceil} < k$ **then**
    Output BinarySearch($a_{\lceil \frac{n}{2} \rceil - 1}), \cdots, a_{n-1}, an$); k;
end
Output NO;

Remark. BinarySearch does work: follows by strong induction.

Theorem 1.2. — *BinarySearch runs in time* $O(\log n)$

Proof. First, note that:

$$T(n) = T(\frac{n}{2}) + c$$

Where the first term is due to the recursion on one problem with half the size and the second term is the constant amount of additional work required. Also, the base case is $T(n) = 1$. Now by adding

the **dummy numbers**, we may assume the n is a power of two: $n = 2^k$. So, we have:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + c \\ &= (T(\frac{n}{4}) + c) + c \\ &= T(\frac{n}{4}) + 2c \\ &= T(\frac{n}{8}) + 3c \\ &= \cdots \\ &= T(\frac{n}{2^k}) + kc \\ &= 1 + kc \\ &= 1 + \log n \cdot c \end{aligned}$$

Thus, $T(n) = O(\log n)$ □

1.3 Master Theorem

Theorem 1.3.1: Master Theorem

If $T(n) = aT(\frac{n}{b}) + O(n^d)$ for constants $a > 0, b > 1, d \geq 0$ then:

$$T(n) = \begin{cases} O(n^d) & a < b^d \ Case\ I \\ O(n^d \log n) & a = b^d \ Case\ II \\ O(n^(\log_b a)) & a > b^d \ Case\ III \end{cases}$$

To prove, we need two following facts:

Lemma 1.3.1. $\sum_{k=0}^{l} r^k = \frac{1-r^(l+1)}{1-r}$ for any $r \neq 1$

Proof. We have:

$$\begin{aligned} (1-r)\sum_{k=0}^{l} r^k &= \sum_{k=0}^{l} r^k - \sum_{k=1}^{l+1} r^k \\ &= r^0 - r^{l+1} \\ &= 1 - r^{l+1} \end{aligned}$$

Divide both sides by $(1-r)$ gives the result. □

Lemma 1.3.2. $x^{\log_b Y} = y^{\log_b X}$

Proof. Observe that, by the power rule of algorithms, $\log_b Z^p = p \cdot \log_b Z$

$$\begin{aligned} \log_b X \cdot \log_b Y &= \log_b Y^{\log_b X} \\ \log_b Y \cdot \log_b X &= \log_b X^{\log_b Y} \\ \log_b Y^{\log_b X} = \log_b X^{\log_b Y} &\Rightarrow x^{\log_b Y} = y^{\log_b X} \end{aligned}$$

□

Now we could prove the Master Theorem:

Proof. Assume n is a power of $b : n = b^l$

$$\begin{aligned} T(n) &= n^d + a(\frac{n}{b})^d + a^2(\frac{n}{b^2})^d + \cdots + a^l(\frac{n}{b^l})^d \\ &= n^d(1 + a(\frac{1}{b})^d + a^2(\frac{1}{b^2})^d + \cdots + a^l(\frac{1}{b^l})^d) \\ &= n^d(1 + \frac{a}{b^d} + (\frac{a}{b^d})^2 + \cdots + (\frac{a}{b^d})^l) \end{aligned}$$

Case 1: $\frac{a}{b^d} < 1$

- Set $r = \frac{a}{b^d}$, then $T(n) = n^d \cdot \sum_{k=0}^{l} r^k$
- Apply Lemma 1, we know that: $\sum_{k=0}^{l} r^k = \frac{1-r^{(l+1)}}{1-r} \leq \frac{1}{1-r} = O(1)$
- Therefore, $T(n) \leq n^d \cdot \frac{1}{1-\frac{a}{b^d}} = n^d \cdot \frac{b^d}{b^d-a} = O(n^d)$

Case 2: $\frac{a}{b^d} = 1$

- $T(n) = n^d(l+1)$
- But $n = b^l$, so $l = \log_b n$
- As b is a constant greater than 1, $T(n) = O(n^d \cdot \log n)$

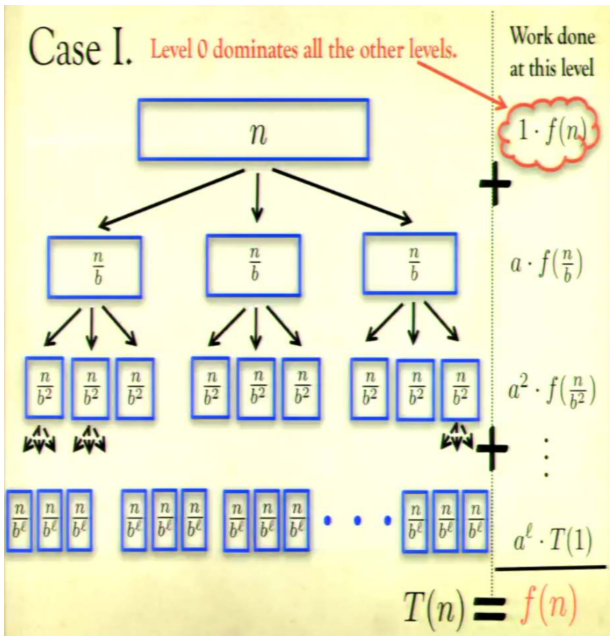
Case 3: $\frac{a}{b^d} > 1$

- Set $r = \frac{a}{b^d}$, then $T(n) = n^d \cdot \sum_{k=0}^{l} r^k$
- Apply Lemma 1 gives: $\sum_{k=0}^{l} = \frac{r^{(l+1)}-1}{r-1} \leq \frac{r^{(l+1)}}{r-1} = O(r^l)$
- $T(n) = O(n^d \cdot r^l)$
- $n^d \cdot r^l = n^d \cdot (\frac{a}{b^d})^l = (\frac{n}{b^l}) \cdot a^l = 1 \cdot a^l = a^{\log_b n} = n^{log_b a}$

□

1.4 The Recursion Tree Method

Master theorem is a special case of the recursion tree method. Specifically, we model the divide and conquer recursive formula by a tree: $T(n) = a \cdot T(\frac{n}{b}) + O(n^d)$; The root node of the tree have a label of n. The root has a children each with label $\frac{n}{b}$. The process stops at the leaves (base case) where $\frac{n}{b^l} = 1$.



1.5 Multiplication

How long does it take to multiply 2 $n-digit$ numbers? Eg. $46 \times 324 = 14904$?

1.5.1 Primary School Long Multiplication

Long Multiplication, it takes n^2 multiplications to multiply two $n-digit$ numbers. Thus, it has running time of $\Omega(n^2)$.

1.5.2 Russian Peasant Multiplication

Algorithm 1.5.1: Russian Peasant Multiplication

Result: Mult(x, y)
if $x = 1$ **then**
  output y;
if x *is odd* **then**
  output $y + Mult(\lfloor \frac{x}{2} \rfloor, 2y)$;
if x *is even* **then**
  output $Mult(\frac{x}{2}, 2y)$;

Remark. There are n iterations, so we add up to n numbers with at most $2n$ digits each. So the runtime of the algorithm is $O(n^2)$.

1.5.3 Divide and Conquer Multiplication

Let $X = 4132, Y = 6703$, So we will have:

$$X = \overbrace{x_n x_{n-1} \cdots x_{\frac{n}{2}+1}}^{X_L} \overbrace{x_{\frac{n}{2}} \cdots x_2 x_1}^{X_R}$$
$$Y = \overbrace{y_n y_{n-1} \cdots y_{\frac{n}{2}+1}}^{Y_L} \overbrace{y_{\frac{n}{2}} \cdots y_2 y_1}^{Y_R}$$

$$X = 10^{\frac{n}{2}} \cdot X_L + X_R$$
$$Y = 10^{\frac{n}{2}} \cdot Y_L + Y_R$$

$$\begin{aligned} X \cdot Y &= (10^{\frac{n}{2}} \cdot X_L + X_R) \cdot (10^{\frac{n}{2}} \cdot Y_L + Y_R) \\ &= \underline{10^n \cdot X_L Y_L} + 10^{\frac{n}{2}} \underbrace{(\underline{X_L Y_R} + \underline{X_R Y_L})}_{simplify\ this\ term} + \underline{X_R Y_R} \end{aligned}$$

So we have 4 multiplication products with $\frac{n}{2}$ digit numbers. Thus $T(n) = 4 \cdot T(\frac{n}{2}) + O(n)$. Here, we may assume that n is a power of 2. By applying the Master Theorem, which in this case ($a = 4, b = 2, d = 1 \Rightarrow$Case III), we get a runtime of $O(n^{\log_2 4}) = O(n^2)$. So far this algorithm is not good. However, we could apply **Gauss multiplication of complex numbers** to make it more efficient.

Gauss Multiplying of Complex Numbers

Normal way of multiplying of complex numbers looks like: $(a + bi)(c + di) = ac - bd + (bc + ad)i$. However, he observes that $(bc + ad) = (a + b)(c + d) - ac - bd$. This implies that we need only one more product to find $(bc + ad)$, namely $(a + b)(c + d)$. In our case, we could replace i by $10^{\frac{n}{2}}$. So instead we have:

$$\begin{aligned} (X_L Y_R + X_R Y_L) &= (X_R + X_L) \cdot (Y_R + Y_L) - X_R Y_R - X_L Y_L \\ &= X_R Y_R + X_L Y_L - \underbrace{(X_R - X_L) \cdot (Y_R - Y_L)}_{1\ multiplication} \end{aligned}$$

So, we end up with 3 products of $\frac{n}{2}$ digit numbers. Now, we have $T(n) = 3 \cdot T(\frac{n}{2}) + O(n)$. By applying Master Theorem ($a = 3, b = 2, d = 1 \Rightarrow$Case III), the runtime now is $O(n^{\log_2 3}) = O(n^{1.59})$.

Remark. Incredibly, we could multiply two n-digit numbers in time $O(n \cdot \log n)$ using a **Fast Fourier Transform**(FFT). FFT is used to multiply two polynomial functions and is used in large applications and is one of the most important numerical algorithm of our time.

1.6 Multiplication of Matrices

How long does it take to multiply 2 $n \times n$ matrices?

1.6.1 High School Matrices Multiplication

The definition of matrix multiplication is that $C = A \cdot B$ for an $n \times m$ matrix A and an $m \times p$ matrix B, then C is an $n \times p$ matrix with entries:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Algorithm 1.6.1: Standard Matrix Multiplication Algorithm

```
Data: Input: Matrix A(n × m) and matrix B(m × p)
Result: C be a matrix of appropriate size (n × p)
for i from 1 to n do
    for j from 1 to p do
        let sum be 0;
        for k from 1 to m do
            set sum ← sum + A_ik × B_kj;
        end
        set C_ij ← sum;
    end
end
Return C;
```

Remark. This algorithm takes $\mathbf{O(nmp)}$. Assume that the matrices are of size $n \times n$, the runtime will be $\mathbf{O(n^3)}$.

1.6.2 Strassen Algorithm for Matrices Multiplication

Assume we have Matrix A *and* B, and matrix C be their product:

$$A = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, B = \begin{bmatrix} E & F \\ G & H \end{bmatrix}, C = AB = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Strassen Algorithm defines:

$$\begin{aligned} S_1 &= (B - D)(G + H) \\ S_2 &= (A + D)(E + H) \\ S_3 &= (A - C)(E + F) \\ S_4 &= (A + B) \cdot H \\ S_5 &= A \cdot (F - H) \\ S_6 &= D \cdot (G - E) \\ S_7 &= (C + D) \cdot E \end{aligned}$$

Now the matrix C becomes:

$$C = \begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$$

Now the multiplying of the matrices involves 7 products instead of 8. The algorithm is now $T(n) = 7 \cdot T(\frac{n}{2}) + O(n^2)$, according to Master Theorem ($a = 7, b = 2, d = 2 \Rightarrow$Case III), the runtime now is $O(n^{\log_2 7}) = O(n^{2.81})$.

Remark. There is a matrix multiplication algorithm that runs in time $O(n^{2.37})$. *Open question: Is matrix multiplication* $O(^{2+\tau})$ *time for any constant* $\tau > 0$.

1.7 Fast Exponentiation

Algorithm 1.7.1: Fast Exponentiation

Data: Input: An unordered list S and the $k-th$ smallest number
Result: FastExp(x, n)
if $n = 1$ **then**
  output x
else
  if n *is even* **then**
    output FastExp$(x, \lfloor \frac{n}{2} \rfloor)^2$
  if n *is odd* **then**
    output $x \cdot$FastExp$(x, \lfloor \frac{n}{2} \rfloor)^2$
end

Remark. We have that $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + 2$, so $T(n) = T(\lfloor \frac{n}{2} \rfloor) + O(1)$ and thus by Master Theorem, the runtime is $O(n^0 \log n) = O(\log n)$.

1.7.1 Domain Transformation

For MergeSort Algorithm, it actually has the recurrence $\widehat{T}(n) = \widehat{T}(\lfloor \frac{n}{2} \rfloor) + \widehat{T}(\lceil \frac{n}{2} \rceil) + cn$. We used dummy entries ($\bar{n}$, the smallest power of 2 greater than n). $\widehat{T}(n) \leq T(\bar{n}) = O(\bar{n} \log \bar{n}) = O(n \log n)$.b Let $T(n) = \widehat{T}(n+2)$. Then:

$$\begin{aligned}
\widehat{T}(n) &\leq \widehat{T}(\frac{n+2}{2} + 1) + c(n+2) \\
&\leq \widehat{T}(\frac{n+2}{2} + 1) + \widehat{c} \\
&= \widehat{T}(\frac{n}{2} + 2) + \widehat{c}n \\
&= T(\frac{n}{2}) + \widehat{c}n
\end{aligned}$$

1.8 The Selection Problem

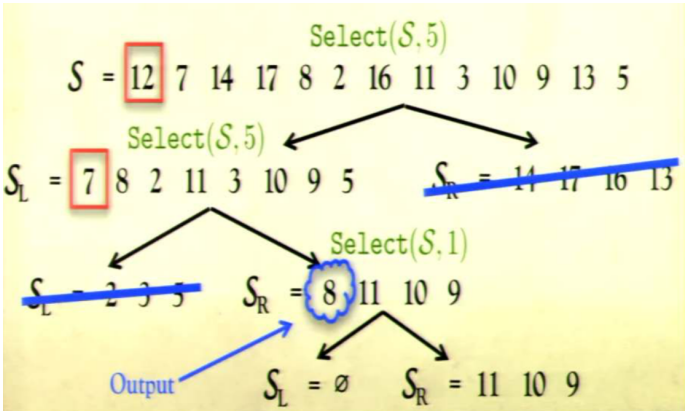
1.8.1 The Median Problem

Suppose that we want to find the median of a set $S = x_1, x_2, \cdots x_n$, we could just sort the list and then output the $\lfloor \lceil n \rceil 2 \rfloor - th$ number. Using the MergeSort Algorithm with runtime $O(n \log n)$. Can we do it in a faster way? We need to investigate the Selection Algorithm.

1.8.2 Naive Selection Problem

Algorithm 1.8.1: Selection Algorithm

Data: An unordered list S, to find the $k-th$ smallest number in list S
Result: Select(S, k)
if $|S| = 1$ **then**
  output x_1;
else
  set $S_L = \{x_i \in S : x_i < x_1\}$ set $S_R = \{x_i \in S \backslash x_1 : x_i \geq x_1\}$;
  if $|S_L| = k - 1$ **then**
    output x_1;
  if $|S_L| > k - 1$ **then**
    output Select(S_L, k);
  if $|S_L| < k - 1$ **then**
    output Select$(S_R, k - 1 - |S_L|)$;
end



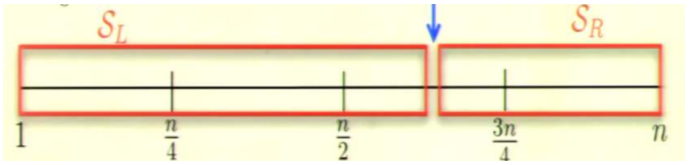
Remark. The naive Selection Algorithm takes $\Omega(n^2)$.

$$\begin{aligned} T(n) &= n-1+T(max\{|S_L|,|S_R|\}) \\ &= n-1+T(n-1) \\ &= (n-1)+(n-2)+T(n-2) \\ &= \cdots \\ &= (n-1)+(n-2)+(n-3)+\cdots+2+1 \\ &= \frac{1}{2}n(n-1) \\ &\Rightarrow T(n) = \Omega(n^2) \end{aligned}$$

This is the worst case scenario. Since we are just choosing the first element as the pivot, $max\{|S_L|,|S_R|\} \approx n$. we can improve the efficiency of the algorithm by choosing a better pivot.

Good vs Bad Pivot

Imagine all the number are in **sorted** order.



With P(the pivot lies in the first and third quartiles) $= \frac{1}{2}$. We say that such a pivot is good. Otherwise the pivot is bad. Note then $max\{|S_L|,|S_R|\} \leq \frac{1}{2}n$. In the worst case, the randomized algorithms will pick the worst pivot. So for the randomized algorithms, we are always interested in the **expected runtime** $\bar{T}(n) = E(T(n))$, NOT the worst case runtime. We have that:

$$\begin{aligned} \bar{T}(n) &\leq \frac{1}{2}\bar{T}(\frac{3n}{4}) + \frac{1}{2}\bar{T}(n) + O(n) \\ &\leq \bar{T}(\frac{3n}{4}) + O(n) \end{aligned}$$

When the first term is due to the probability of a good pivot and the second term is due to the probability of a bad pivot. By applying the Master Theorem($a = 1, b = \frac{3}{4}, c = 1$), $\bar{T}(n) = O(n)$.

1.8.3 Deterministic Algorithm

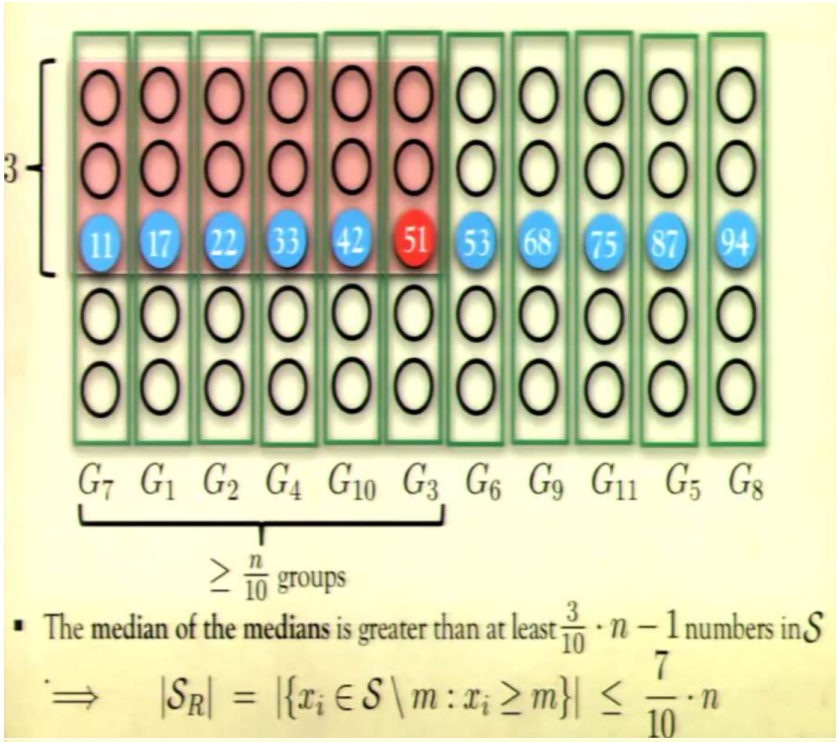
Is there a linear time deterministic algorithm? To do this, what we need is a deterministic method to find a good pivot. The idea is to find the *Median of the Medians*.

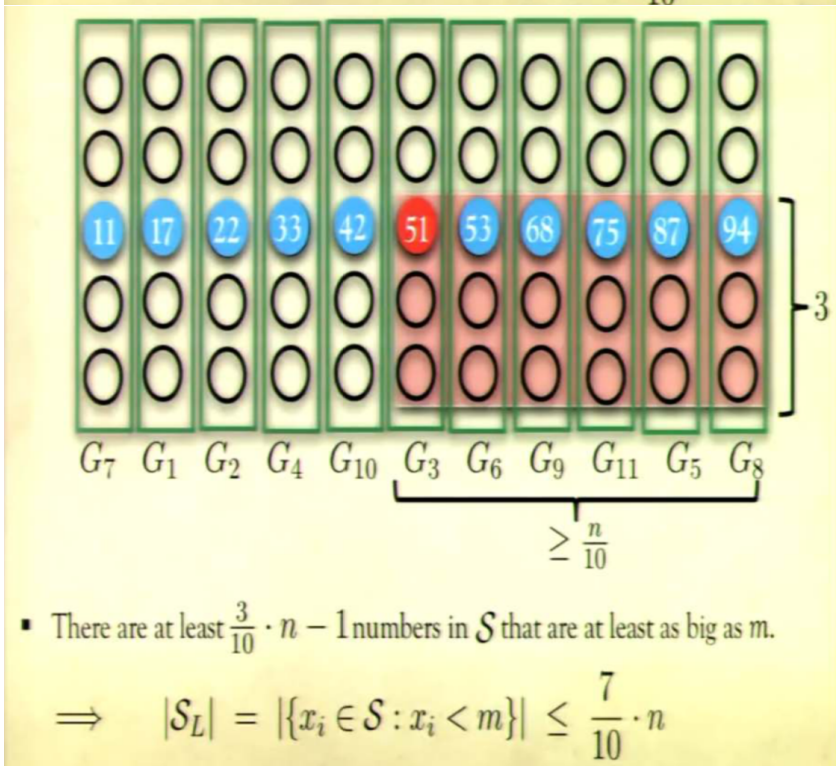
Median of the Medians

Divide S into groups of cardinality 5:

$$G_1 = \{x_1, \cdots, x_5\}, G_2 = \{x_6, \cdots, x_{10}\}, G_{\frac{n}{5}} = \{x_{n-4}, \cdots, x_n\}$$

Now, sort each group and let z_i be the medians of the group G_i. Let m be the median of $Z = \{z_1, \cdots, z_{\frac{n}{5}}\}$. Reorder the groups by their median values.





So,using m as a pivot, we have: ,$\max\{|S_L|, |S_R|\} \leq \frac{7}{10}n$. So, the median of the medians is a good pivot. but how do we actually find the median of the medians? Well, we just use the same deterministic algorithm:

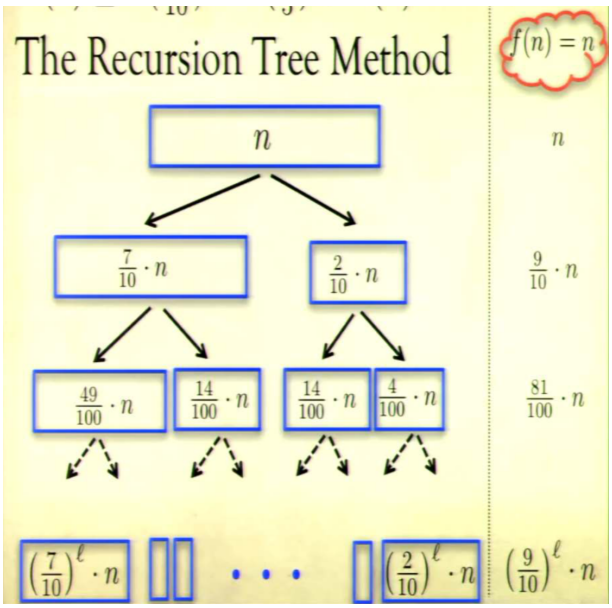
Algorithm 1.8.2: Deterministic Selection Algorithm

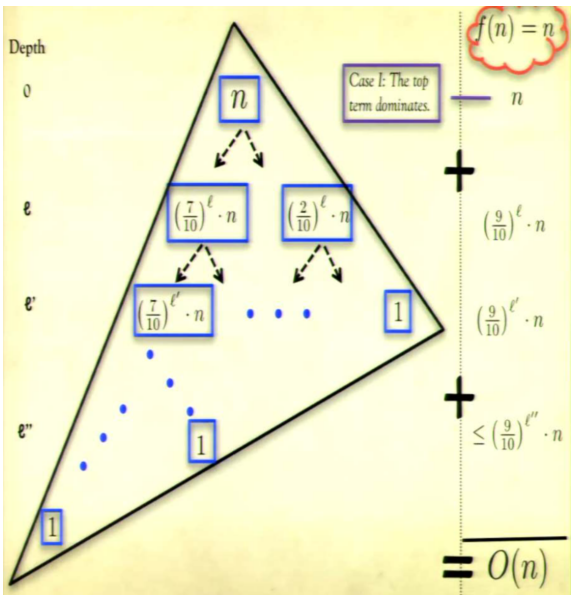
Data: Input: An unordered list S and the $k-th$ smallest number
Result: DetSelect(x, k)
if $|S| = 1$ **then**
  output x_1
else
  Partition S into $\lceil \frac{n}{5} \rceil$groups of 5.;
  for $j = \{1, 2, 3, \cdots, \lceil \frac{n}{5} \rceil\}$ **do**
    Let z_j be the median of Group G_i;
  end
  Let $Z = \{z_1, z_2, \cdots, z_{\lceil \frac{n}{5} \rceil}\}$;
  Set $m \leftarrow$ DetSelect$(Z, \lceil \frac{n}{10} \rceil)$;
  Set $S_L = \{x_i \in S : x_i < m\}$ set $S_R = \{x_i \in S \backslash m : x_i \geq m\}$;
  if $|S_L| = k - 1$ **then**
    output m;
  if $|S_L| > k - 1$ **then**
    output DetSelect(S_L, k);
  if $|S_L| < k - 1$ **then**
    output DetSelect$(S_R, k - 1 - |S_L|)$;
end

Remark. By using m as a pivot, we have: $max\{|S_L|, |S_R|\} \leq \frac{7}{10} \cdot n$. The recursive formula for the running time is then:

$$T(n) \leq T(\frac{7n}{10}) + T(\frac{n}{5}) + O(n)$$

The first term is due to that pivoting on the median of the medians gives a significantly smaller subproblem. The second term is from finding the median of the medians. The last term is from breaking in groups of 5 and find the median of each group. Pivoting on the median of medians. But this does not fit into the Master Theorem. We can instead simply apply the recursion tree method as shown below. The resulting runtime of the algorithm is $T(n) = O(n)$.





1.9 Finding the Closest Pair of Points in the Plane

Given n points $P = \{P_1, \cdots, P_n\}$ in the XY plane, we would like to find the closest pair of points.

1.9.1 Exhaustive Search

The simplest algorithm to try is Exhaustive Search. The idea is to calculate the distance between every pair of points and then output the pair with the shortest pairwise distance. Since there are n points, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of points and thus the runtime is $O(n^2)$. Is there anything faster?

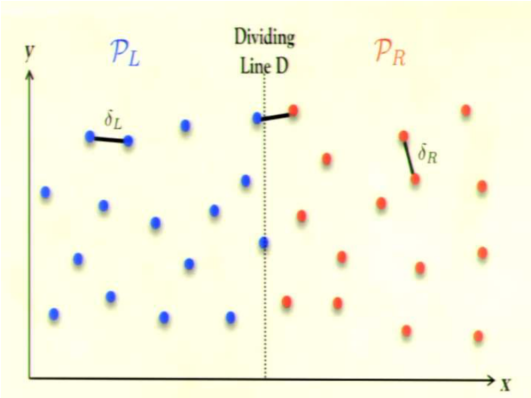
1.9.2 1D Case and a Naive Approach

It will be informative to first examine our problem in one dimension. In one dimension, the closest pair of points must be adjacent in their $x - ordering$. Thus, we need to calculate only $n-1$ pairwise distances, resulting in a runtime of $O(n)$ if the points are sorted in the x coordinate and runtime of $O(n \log n + n) = O(n \log n)$ otherwise.
What happens if we apply this idea in 2 dimensions? Let x coordinate (called the $x - ordering$) and then order the points by their y coordinate (called the $y - ordering$). Then, find the pair of points closest in their $x - coordinate$. And, find the pair of points closest in their $y - coordinate$. From these 2 pairs, output the pair that are closest together. Obviously, this algorithm does not work at all.

1.9.3 Divide and Conquer Algorithm with a Trick

Let's find an algorithm to find the closest pair of points in the plane.

First, partition the points into 2 groups of cardinality $\frac{n}{2}$. We do this via the $x - ordering$, using a dividing line D. Select D to pass through the point with the median $x - coordinate$. We can find the median from the previous section in $O(n)$. Now let's recursively search for the closest pairs in P_L and P_R.



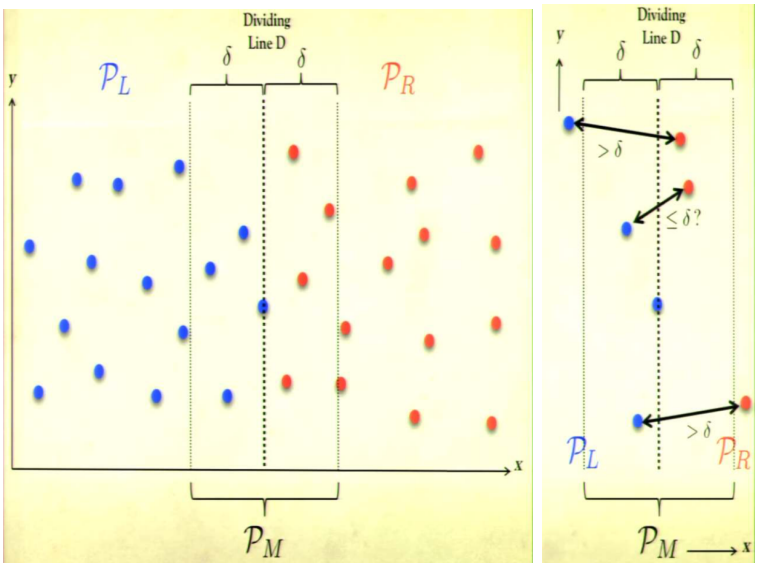
But there is a third possibility, the closest point could have one point in P_L and the other in P_R. Thus, after calculating δ_L and δ_R, we must check that there is no better solution between a point in P_L and a point P_R. We can have a recursive algorithm by now:

1. Find the point q with the median $x - coordinate$.
2. Partition P into P_L and P_R using point q.
3. Recursively find the closest pairs of points in P_L and P_R.
4. Find the closest pair with one point in P_L and one point in P_R.
5. Amongst the 3 pairs found, output the closest pair.

So the recursive formula for the running time is: $T(n) = 2 \cdot T(\frac{n}{2}) + O(n^2)$. For the term $O(n^2)$, there are $\frac{n}{2}$ points in P_L and $\frac{n}{2}$ in P_R, so there are $\frac{n^2}{4}$ pairs of points. Thus, we have $a = 2, b = 2, d = 2$. This is Case I of the Master Theorem. The runtime for the algorithm is $O(n^d) = O(n^2)$.

The Bottleneck Step

So the bottleneck operation is Step 4 (Find the closest pair with one point in P_L and one point in P_R). We can apply a trick to tackle the bottleneck. So it takes far too long to measure the distance between every point in P_L and every point in P_R. But by solving the subproblems of P_L and P_R, we know the **minimum pairwise distance** is at most: $\delta = min\{\delta_L, \delta_R\}$. The trick is to use the value of the δ (which we have calculated by recursion) to reduce the number of distances we measure between P_L and P_R. The key observation is that to find a better solution than δ the two points in P_L and P_R must be very close to the **dividing line D**.



Lemma 1.9.1. Take $p_i \in P_L$ and $p_j \in P_R$. If $d(p_i, p_j) \leq \delta$ then $(p_i, p_j) \subseteq P_M$.

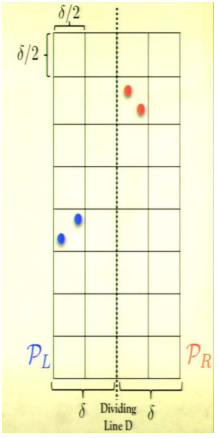
Proof. WLOG take $p_i \notin P_M$. Then, $d(p_i, p_j) > \delta$. □

A Modified Recursive Algorithm

This include the following **fine-tuned** divide and conquer algorithm.

1. Find the point q with the **median** x-coordinate.
2. Partition P into P_L and P_R using point q.
3. Recursively find the closest pairs of points in P_L and P_R.
4. Find the closest pair of points in P_M.
5. Amongst the three pairs found, output the **closest** pair.

However, P_M may contain all (or most) of the points. It may still take $\Omega(n^2)$ to measure the distance between the points in $P_L \cap P_M$ and the points in $P_R \cap P_M$. The points in P_M cover a narrow band of the x-axis. As we have a target distance of δ.



Consider the area of the plane within distance δ of the line D. Divide the area up into small squares of width $\frac{\delta}{2}$ as shown. We can claim that no two points of P lies in the same square. To prove this claim we will use the following observation: Each square lies entirely on one side of D.

Lemma 1.9.2. No two points of P lies in the same square.

Proof. WLOG Take two points in a square in P_L. Their pairwise distance is $\leq \sqrt{(\frac{\delta}{2})^2 + (\frac{\delta}{2})^2} = \frac{\delta}{\sqrt{2}} < \delta$. This contradicts the fact that the smallest pairwise distance in P_L is at least δ. □

Theorem 1.3. — *Take $p_i \in P_L$ and $p_j \in P_R$. If $d(p_i, p_j) \leq \delta$ then p_i and p_j have at most 10 points between them in the y-order of P_M.*

Proof. WLOG p_i is below p_j in the y-order. Then p_j is in the same row of squares as p_i or in the next two higher rows. If not, $d(p_i, p_j) > \delta$, and there is a contradiction. But, by the claim, there is at most one point in each square. ⇒There are at most 10 points between p_i and p_j in the y-order of P_M. □

So back to the 1-D case. This means the basic idea of the 1-D algorithm will work here! To find the closest pair in P_M, we first sort the points by their y-coordinate. Then, rather than finding the distance between points that are 1-apart in the y-order, we find the distance for all pairs up to 11 places apart. Thus, we calculate less than $11n$ pairwise distance. After doing so, either we find a pair of points at distance less than δ. Or we conclude that no such pair of points exists. So given $\delta = min\{\delta_L, \delta_R\}$, we can check if there is a pair of points between P_L and P_R that are closer than δ in time $O(n)$.

An Enhanced Modified Recursive Algorithm

Finally, this will give us very fast divide and conquer algorithm.

1. Find the point q with the **median** x-coordinate.
2. Partition P into P_L and P_R using point q.
3. Recursively find the closest pair of pairs of points in p_R and P_R.
4. Find the closest pair of points in p_M using the **enhanced** 1-D algorithm.
5. Amongst the three pairs found, output the closest pair.

So the recursive formula for the running time is: $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$. (The last term is due to finding median with respect to x-coordinate; Partition into P_L and P_M; Find P_M and apply 1-D algorithm on P_M.) The runtime of the algorithm is $O(n^d \cdot \log n) = O(n \cdot \log n)$.

Validity

As usual, the correctness of the algorithm follows by **strong induction**. For the base case, we can find the closest pair in constant time by exhaustive search when there are a constant number of points left. That the **inductive step** is correct follows by our discussion in the lecture.

Computational Geometry

The closest pair of points problem was a fundamental question in the field of **computational geometry**. Computational geometry has numerous applications in: *Computer Graphics, Computer vision, Geographic Information System, Computer aided Design, Circuit Design, Robotics and Motion Planning, etc.*

Part II

Greedy Algorithms

2 | Scheduling

2.1 A Brief Overview

The simplest class of algorithms are greedy algorithms. They make a locally optimal (or myopic) choice at each step. Greedy algorithms have several nice properties: *Fast, Simple, Easy to Code.* Unfortunately, they are also usually *rubbish.* So for this topic the goal is to understand the basic greedy algorithmic techniques and on which problems do these techniques work.

2.2 Task Scheduling

A firm receive job orders from n customers. The firm can do only one task at a time. The job of customer i will take the firm t_i units of time to complete. Each customer i wants their job completed as early as possible(and will pay accordingly). Assume the firm process the jobs in the order $\{1, 2, 3 \cdots, n\}$. Then the waiting time of customer i will be: $w_l = \sum_{i=1}^{l} t_i$. So to maximize the profits the objective of the firm is to minimizing the sum of the waiting times:

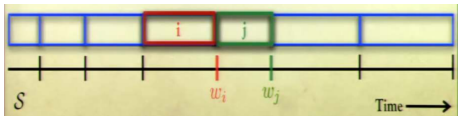
$$\sum_{w_l}^{n} w_l = \sum_{l=1}^{n} \sum_{i=1}^{l} t_i$$

Here is a greedy algorithm that solves this problem optimally.
Greedy Task Scheduling Algorithm:
Sort the jobs by length $t_1 \leq t_2 \leq t_3 \leq \cdots \leq t_n$.
Schedule the jobs in the order $\{1, 2, 3, \cdots, n\}$.

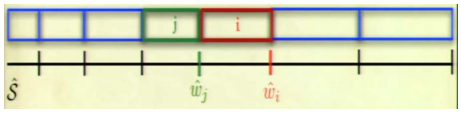
Theorem 2.1. — *The greedy algorithm outputs an optimal schedule.*

Proof. Let the greedy algorithm schedule in the order $\{1, 2, \cdots, n\}$. Assume there is a better schedule S. Then there must be a pair of jobs i and j such that:

- Job i is scheduled immediately before job by schedule S.
- Job i is longer than job j: $t_i > t_j$.



But then exchanging the order of jobs i and j gives a better solution $\hat{S}$.



Observe the waiting time of every other job remains the same. $\hat{w_k} = w_k, \forall k \neq i, j$. But clearly: $\hat{w_i} + \hat{w_j} < w_i + w_j$. This contradicts the assumption that S was an optimal schedule. □

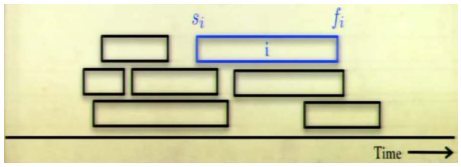
Remark. In order to optimize the scheduling process, we need only sort n time lengths.

$$\Rightarrow \text{Runtime} = O(n \cdot \log n).$$

So this algorithm is efficient.

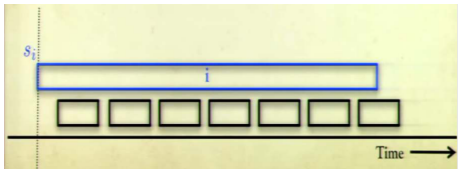
2.3 Class Scheduling

There is one classroom. There is a set $I = \{1, 2, 3, \cdots, n\}$ of classes that request room. Class i has a start time S_i and a finish time f_i. The objective is to book as many classes into the room as possible. However, we cannot book two classes that need to use the room at exactly the same time. This problem is often called the **Interval Selection Problem**, as we can plot the classes as intervals between their start and finish time.



2.3.1 First Start

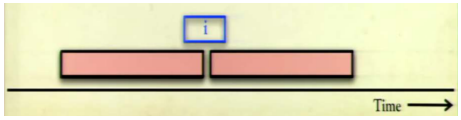
Select the class that starts earliest, and iterate on the remaining classes that do not conflict with this selection.



This doesn't work since the first class might be the longest.

2.3.2 Shortest-Duration

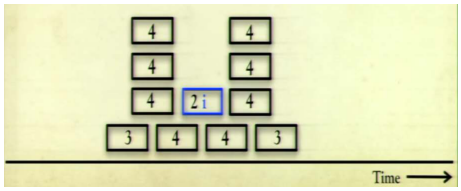
Select the class of shortest duration, and iterate on the remaining classes that do not conflict with this selection.



This doesn't work because it might be in an unlucky position.

2.3.3 Minimum-Conflict

Select the class that conflicts with the fewest number of classes, and then iterate.



Here the optimal solution has 4 classes, but we chose the configuration that has only 3, we chose i since i conflicts with only 2, whereas the red ones all conflict with 3 or 4. Next time we iterate, we just have two stacks of 3, so we can choose any from the left or right, and were stukc with having at most 3 classes in our solution

2.3.4 Last Start

Select the class that starts last, and iterate on the classes that do not conflict with this selection. It outputs the optimal schedule. This algorithm is symmetric to the following greedy algorithm: **First Finish**. Select the class that finishes first, and iterate on the classes that do not conflict with this selection.

First-Finish Greedy Scheduling

Algorithm 2.3.1: First Finish Greedy Scheduling

Result: FirstFinish(I)
Let class 1 be the class with the earliest finish time;
let X be the set of classes that clash with class 1;
Output {1}∪ FirstFinish(I\X);

First-Finish Greedy Scheduling (Revisited)

Algorithm 2.3.2: First Finish Greedy Scheduling (revisited)

Result: FirstFinish(I)
Sort the classes by finish times $f_1 \leq f_2 \leq \cdots \leq f_n$;
Initialize the set of selected classes: $S = \emptyset$;
Initialize the set of undecided class requests: $I = \{1, 2, 3, \cdots, n\}$;
while $I \neq \emptyset$ **do**
  Let i be the lowest index class in I;
  Set $S \leftarrow S \cup \{i\}$ and $I \leftarrow I \setminus \{i\}$;
  for *Each class* $j \in I$ **do**
    if $s_j < f_i$ **then**
      Set $I \leftarrow I \setminus \{j\}$;
  end
end
Output S;

Remark. There are at most n iterations. It takes $O(n)$ time to find class that finishes earliest in each iteration.

$$\Rightarrow Runtime = O(n^2)$$

But there is a very crude analysis. With a more subtle implementation and analysis we can obtain a running time of $O(n \cdot \log n)$.

2.4 The Shortest Path Algorithm

Part III

Dynamic Programming

3 | Dynamic Programming

Part IV

Network Problem

4 | Max-Min Cut problem